

Informática

Python: Conceptos Básicos

Josefina Sierra Santibáñez

jsierra@cs.upc.edu

adaptación de las transparencias de

Joaquim Gabarro

gabarro@cs.upc.edu

Computer Science

Universitat Politècnica de Catalunya

Contenidos

1. Tipos de datos básicos, literales, variables, expresiones y operadores aritméticos, relacionales y lógicos.
2. Instrucciones básicas: asignación (`=`), entrada (`input`) y salida (`output`).
3. Funciones: uso de bibliotecas (libraries), especificación, parametrización y valor de retorno.
4. Composición secuencial, alternativa (`if-elif-else`) e iterativa (`for`, `while`).
5. Tratamiento de secuencias: esquemas de recorrido y de búsqueda.
6. Estructuras de datos: strings, listas, clases/objetos y diccionarios.
7. Algoritmos fundamentales de búsqueda, aproximación numérica y ordenación.

Se utilizará un subconjunto del lenguaje de programación `Python 3`. Si bien el curso pretende profundizar en la `resolución de problemas algorítmicos` y en la `construcción estructurada de programas`.

Método de Evaluación

- ▶ **Lab_1** (examen parcial de laboratorio): sesión 6 (ordenador 1h)
- ▶ **Lab_2** (examen parcial de laboratorio): sesión 10 (ordenador 1h)
- ▶ **Lab_3** (examen parcial de laboratorio): sesión 14 (ordenador 2h)
- ▶ **Teo** (examen final de teoría): Enero (papel 2h)

$$\text{Nota global} = 0,20 \cdot \text{Lab}_1 + 0,25 \cdot \text{Lab}_2 + 0,25 \cdot \text{Lab}_3 + 0,30 \cdot \text{Teo}$$

Tipos de datos y Expresiones

Los programas manipulan **datos**. En este curso veremos distintos **tipos de datos** que permiten representar números, texto y valores de verdad, entre otros.

Todos los datos deben representarse en el ordenador utilizando un formato digital. Los datos de tipos diferentes se representan de forma distinta.

Una **expresión** es un fragmento del código de un programa que produce o calcula un nuevo dato.

El tipo más simple de expresión es un **literal**. Un literal es el nombre de un valor determinado. Por ejemplo, **3.9** y **12** son literales numéricos, **True** y **False** son literales de tipo **bool** (i.e., valores de verdad o Booleanos) y la frase **''Bon dia''** es un literal de tipo textual (i.e., una **cadena de caracteres** o **string**).

Tipos de datos y Expresiones

El tipo de datos que se utiliza para representar números enteros es `int` (e.g., el literal 12 es de tipo `int`).

El tipo de datos que se utiliza para representar números con parte decimal es `float` (e.g., el literal 3.9 es de tipo `float`).

El tipo de datos que se utiliza para representar datos de tipo textual se denomina `string`. Una `string` es una secuencia de caracteres imprimibles. Un `literal de tipo string` se representa en Python encerrando la secuencia de caracteres entre comillas (" "). Por ejemplo, "Introduce un numero mayor que 0: ".

Las comillas propiamente dichas no forman parte de una `string`, son simplemente un mecanismo que permite indicar a Python que deseamos crear una `string`.

Tipos de datos `int` y `float`

Los **tipos de datos numéricos** son representaciones de tipos de números (e.g., los enteros o los racionales). Sin embargo, es importante tener en cuenta que las representaciones de los números en el ordenador (los tipos de datos numéricos) no siempre se comportan como los números de las matemáticas que representan.

La **Unidad de Procesamiento Central (CPU)** del ordenador puede realizar operaciones aritméticas básicas (como sumar y multiplicar) con las representaciones internas de los números en el ordenador.

El tipo de dato `int` se almacena utilizando **una representación binaria de longitud fija** dentro del ordenador.

La memoria del ordenador está compuesta de “conmutadores eléctricos”, cada uno de los cuales puede estar en uno de dos estados: encendido o apagado. Un conmutador representa un **dígito binario** o **bit** de información.

Tipos de datos `int` y `float`

Un bit puede codificar dos posibilidades, normalmente representadas con los números 0 (para apagado) y 1 (para encendido).

Una secuencia de bits permite representar más posibilidades. n bits permiten representar 2^n valores distintos.

El número de bits que utiliza un ordenador concreto para representar un dato de tipo `int` depende del diseño de su CPU. Los ordenadores personales (PCs) actuales utilizan 32 o 64 bits.

En una CPU de 32 bits, la representación del tipo `int` permite almacenar 2^{32} valores diferentes. Estos valores se centran en el 0, para representar tanto enteros positivos como negativos. Por tanto, el rango de números enteros que se pueden representar en el tipo `int` con el que opera la CPU es $[-2^{31}, 2^{31} - 1]$.

Tipos de datos int y float

Los valores de tipo `float` (i.e., de tipo `punto flotante`) son siempre aproximaciones de los números racionales o reales de las matemáticas que representan.

La notación científica es una forma de escribir números largos de forma concisa. Los números en notación científica toman la forma siguiente: `parte_significativa` $\times 10^{\text{exponente}}$. Por ejemplo, el número 12000 en notación científica es $1,2 \times 10^4$, 1,2 es la parte significativa y 4 es el exponente.

Por convenio, los números en notación científica se escriben con un dígito delante del punto decimal y el resto de los dígitos detrás. En Python, se usa la letra `e` para representar la parte *10 elevado a*. Por ejemplo, $1,2 \times 10^4$ se escribe `1,2e4`.

Tipos de datos int y float

Los dígitos de la parte significativa (la que está delante de e) se denominan *dígitos significativos*. El número de dígitos significativos define la **precisión** de un número. Cuantos más dígitos significativos tiene, más preciso es un número.

Los números de tipo punto flotante sólo pueden almacenar un número determinado de dígitos significativos, los demás se pierden. La precisión de un número de punto flotante indica la cantidad de dígitos significativos que puede representar sin pérdida de información.

Tipos de datos int y float

Los números de tipo **punto flotante** (es decir, los que tienen una parte decimal) se representan en el ordenador mediante un par de números enteros binarios de longitud fija. Un entero representa la parte significativa (secuencia de dígitos del valor), y el otro entero representa el valor del exponente, que indica donde termina la parte entera y donde comienza la parte fraccional.

El número de dígitos de precisión que tiene una variable de tipo punto flotante depende de su tamaño (el número de bits que se usan para representarla en la memoria del ordenador) y del valor particular almacenado (algunos valores tienen menos precisión que otros).

float

Size: 8 bytes (64 bits)

Range: $\pm 2,23 \times 10^{-308}$ to $\pm 1,80 \times 10^{308}$

Precision: 15-18 significant digits, typically 16

Tipos de datos int y float

Otra de las razones por las que los números de tipo punto flotante pueden ser problemáticos es debido a la diferencia entre el sistema binario (en el que se representan los datos en la memoria del ordenador) y el sistema decimal (que utilizamos nosotros para escribirlos).

En el sistema decimal $1/10$ se representa como 0.1. Pero en binario se representa mediante la secuencia infinita 0.00011001100110011... Por este motivo, cuando asignamos 0.1 a un número de tipo punto flotante, internamente se almacena como 0.1000000001, porque es necesario truncar su aproximación debido a la memoria limitada de la representación interna.

Este tipo de error, se conoce como **error de redondeo**. Las operaciones matemáticas (como la suma y la multiplicación) tienden a aumentar los errores de redondeo.

Tipos de datos int y float

Afortunadamente Python permite representar valores grandes exactos de una manera mejor. El tipo de dato `int` en Python no tiene un tamaño fijo. Se expande para acomodar el valor que toma. El único límite a dicha expansión es la cantidad de memoria de la que dispone el ordenador.

Cuando el valor entero es pequeño, Python puede utilizar la representación de `int` y las operaciones aritméticas de la CPU (i.e., una representación de 32 bits).

Cuando el valor entero es mayor, Python convierte el valor automáticamente a una representación que utiliza más bits.

Conversiones de tipos de datos int y float

Python proporciona funciones que permiten realizar conversiones de tipo de datos de forma explícita como `int(x)`, `float(x)` y `str(x)`.

Existe además una función denominada `round(x)` que redondea un número de tipo `float` al número de tipo `int` más cercano a él.

Finalmente, también se puede utilizar la variante `round(x,n)` para redondear un número de tipo `float` `x` al número de tipo `float` más cercano a él con `n` cifras decimales.

Tipos de datos y Expresiones

El tipo de datos que se utiliza para representar datos de tipo textual se denomina **string**. Una string es una secuencia de caracteres imprimibles. Un **literal de tipo string** se representa en Python encerrando la secuencia de caracteres entre comillas (" "), por ejemplo "Introduce un número mayor que 0:". Estos literales producen las strings que contienen las secuencias de caracteres entrecomilladas. Las comillas propiamente dichas no forman parte de una string, son simplemente un mecanismo que permite indicar a Python que deseamos crear una string.

El proceso de calcular el dato concreto que representa una expresión se denomina **evaluación**.

Cuando escribimos una expresión en la línea de comandos, el intérprete de Python evalúa dicha expresión y escribe a continuación una representación textual del resultado.

Observad que cuando Python muestra el valor de una string, escribe la secuencia de caracteres entre comillas sencillas (' ').

Tipos de datos y Expresiones

Las **variables** se utilizan para asignar nombres a ciertos datos. Técnicamente esos nombres se denominan **identificadores**. La diferencia entre un literal y una variable es que el literal siempre representa el mismo dato, pero la variable puede representar distintos datos durante la ejecución de un programa.

Un identificador puede comenzar con una letra o con el signo de subrayado (i.e., el carácter `_`). El resto de los caracteres de un identificador pueden ser letras, dígitos o signos de subrayado. En los identificadores el uso de mayúsculas y minúsculas es importante, por ejemplo, los identificadores `g1` y `G1` son diferentes. Es preciso asignar un valor a una variable antes de usarla en una expresión. Es decir, toda variable debe ser inicializada antes de usarla.

Algunos identificadores forman parte del lenguaje de programación Python. Estos identificadores se denominan **palabras reservadas** o **palabras clave** (i.e. **keywords**) y no pueden utilizarse como identificadores ordinarios.

Tipos de datos y Expresiones

Es preciso asignar un valor a una variable antes de usarla en una expresión.

Una expresión también puede ser una **variable**. El resultado de evaluar una expresión que consiste en una sola variable es el dato asignado a dicha variable.

El resto de las **expresiones se construyen combinando expresiones más sencillas mediante operadores**.

Python proporciona los operadores **+**, **-**, *****, **/** y ****** para datos numéricos. Los operadores matemáticos obedecen las mismas reglas de precedencia y asociatividad habituales en matemáticas, incluyendo el uso de paréntesis para modificar el orden de evaluación.

Python incluye también el operador **+** para datos de tipo `string`, que permite **concatenar** varias cadenas de caracteres.

También están definidos los operadores de división entera **//** (cociente entero) y **%** (resto) para el tipo de datos `int`.

Variables y tipos: euros y nombre

Una **variable** toma un **valores** y tiene **tipo**.

Es preciso asignar un valor a una variable antes de usarla en una expresión. Es decir, toda variable debe ser inicializada antes de usarla.

Ejemplo: variable **euros** de tipo integer: **int**

```
>>> euros = 500
>>> euros
500
>>> euros = euros + 50
>>> euros
550
>>> type(euros)
<class 'int'>
```

A la siguiente **intrucción** se le llama **asignación**.

```
>>> euros = euros + 50
```

Leer: **euros** toma por **valor** el que tenía antes más 50.

Tipo de dato

El tipo de un objeto determina los valores que puede tomar y las operaciones que se pueden realizar con dicho objeto.

Python proporciona una función especial denominada `type` que devuelve el `tipo de dato` (o `clase`) de cualquier valor.

Ejemplo: variable `nombre` de tipo `string` (o cadena de caracteres):
`str`.

```
>>> nombre = "paco"  
>>> type(nombre)  
<class 'str'>  
>>> apellido = "perez"  
>>> nombre = nombre + " " + apellido  
>>> nombre  
'paco perez'  
>>> nombre = "pepe"  
>>> nombres = 3*nombre + " " + "adios"  
>>> nombres  
'pepepepepepepe adios'
```

Ejemplo: tipo `float`

```
>>> fraccion = 1/3  
>>> type(fraccion)  
<class 'float'>
```

Expresiones Booleanas

Una **condición** es una expresión cuyo valor es un dato de tipo **bool** (i.e. **Booleano** o **valor de verdad**). Las condiciones se denominan también **expresiones booleanas**.

Una **condición simple** tiene la forma siguiente

`<expr> <relop> <expr>`

`<relop>` es un **operador relacional**. Los siguientes operadores son relacionales.

`<`, `<=`, `>`, `>=`, `==`, `!=`

Estos operadores permiten comparar números o strings (cadenas de caracteres) utilizando el **orden lexicográfico**.

El resultado de evaluar una expresión booleana puede ser **True** (si la condición es cierta) o **False** (si la condición es falsa).

Expresiones Booleanas

A menudo las condiciones simples que se pueden construir utilizando operadores relacionales no son suficientemente expresivas. Los **operadores Booleanos and, or, not** permiten construir expresiones Booleanas complejas a partir de expresiones Booleanas más sencillas.

Los operadores Booleanos **and** y **or** permiten combinar dos expresiones Booleanas. Su resultado es un valor Booleano.

`<expr> and <expr>`

`<expr> or <expr>`

La **conjunción (and)** de dos expresiones Booleanas es cierta exactamente cuando las dos expresiones son ciertas (ver **tablas de verdad**).

La **disyunción (or)** de dos expresiones Booleanas es cierta cuando alguna de las dos expresiones son ciertas (ver **tablas de verdad**). La disyunción es falsa sólomente cuando ambas expresiones son falsas.

Expresiones Booleanas

El operador `not` calcula el opuesto de una expresión Booleana. Es un operador unario de la forma

`not <expr >`

El resultado de evaluar `not <expr >` es True (cierto) si el resultado de evaluar `<expr >` es False (falso); y viceversa. El resultado de evaluar `not <expr >` es False (falso) si el resultado de evaluar `<expr >` es True (cierto).

Utilizando operadores Booleanos es posible construir expresiones Booleanas muy complejas. Como en el caso de los operadores aritméticos, el resultado de evaluar una expresión Booleana compleja depende de las reglas de prioridad de los operadores Booleanos.

El orden de prioridad de más alta a más baja para los operadores Booleanos es el siguiente: `not`, `and`, `or`.

En cualquier caso es recomendable utilizar paréntesis en expresiones Booleanas complejas para facilitar su comprensión.

Expresiones Booleanas

En la evaluación de una expresión Booleana de tipo `or`, la expresión de la parte izquierda del operador `or` se evalúa en primer lugar. Si el resultado es `True`, Python devuelve el resultado `True` como resultado de evaluar la disyunción, y no evalúa la expresión de la parte derecha del operador `or`.

Este tipo de evaluación se denomina **evaluación en corto-circuito**.

De manera similar, en la evaluación de una expresión Booleana de tipo `and`, la expresión de la parte izquierda del operador `and` se evalúa en primer lugar. Si el resultado es `False`, Python devuelve el resultado `False` como resultado de la evaluación de la conjunción y no evalúa la expresión de la parte derecha del operador `and`.

Operadores

- ▶ Operadores aritméticos : +, -, *, /, // (división entera), % (resto de división entera), ** (exponenciación)
- ▶ Operadores de concatenación de strings: +, *
- ▶ Operadores relacionales (comparaciones): ==, !=, <, <=, >, >=
- ▶ Operadores relacionales (substring de un string): in, not in
- ▶ Operadores lógicos: not, and, or

Se pueden concatenar:

```
>>> 3<5<7<9
```

```
True
```

```
>>> 3+5+7+9
```

```
24
```


Prioridad

- ▶ Python tiene reglas que especifican el orden en que los operadores en una expresión se evalúan.
- ▶ Por ejemplo, la multiplicación y la división tienen mayor prioridad que la suma y la resta.
- ▶ Ejemplos:

```
>>> 1+2*3
```

```
7
```

```
>>> 1+2*3<8
```

```
True
```

Operator precedence

From highest precedence (most binding) to lowest precedence (least binding)

Order	Operators	Grouping
1	** (exponentiation)	right to left)
2	+ , - (positive, negative,)	-
3	* , / , // , % , (multiplication, division, floor division, remainder)	left to right
4	+ , - (addition and subtraction)	left to right
5	Comparisons, membership and tests in , not in , is , is not , < , <= , > , >= , != , ==	left to right
6	not	-
7	and	left to right
8	or	left to right

Example:

```
>>> 4*5-2
```

```
18
```

- ▶ Con paréntesis (redundantes si se aplican las reglas de prioridad) las expresiones anteriores corresponden:

```
>>> 1+(2*3)
```

```
7
```

```
>>> (1+(2*3))>5
```

```
True
```

- ▶ Recomendación de Python: ante la duda pon paréntesis.
- ▶ Las reglas de prioridad se anulan al poner paréntesis explícitos.

```
>>> 1+2*3
```

```
7
```

```
>>> (1+2)*3
```

```
9
```

Lógica booleana

Type `bool = {True, False}`

```
>>> True
True
>>> False
False
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

Operadores relacionales de comparación

<code>==</code> igual	<code>!=</code> diferente
<code><</code> menor	<code><=</code> menor o igual
<code>></code> mayor	<code>>=</code> mayor o igual

Os sonará el concepto de Tablas de Verdad.

Las tablas de verdad son parte de la lógica booleana.

Tablas de verdad

not (negation), and (conjunction), or (disjunction)

x	y	not x	not y	x and y	x or y
True	True	False	False	True	True
True	False	False	True	False	True
False	True	True	False	False	True
False	False	True	True	False	False

```
>>> x = True
>>> y = False
>>> not x
False
>>> x and y
False
>>> x or y
True
```

In many math books: not x is written \bar{x} ,
 x and y is $x \wedge y$ and x or y is $x \vee y$.

Example

```
>>> 1<=3 and not 6>3  
False
```

Solución:

```
1<=3 and not 6>3  
-- add parenthesis to clarify  
= ((1<=3) and (not (6>3)))  
-- evaluate arithmetic expressions  
= (True and (not (True)))  
-- evaluate logic expression  
= (True and False)  
= False
```

De Morgan's laws

```
(not (x and y)) == ((not x) or (not y))
```

```
(not (x or y)) == ((not x) and (not y))
```

Also written $\overline{x \wedge y} = \bar{x} \vee \bar{y}$ and $\overline{x \vee y} = \bar{x} \wedge \bar{y}$

For instance:

```
>>> x = True
```

```
>>> y = False
```

```
>>> (not (x and y)) == ((not x) or (not y))
```

```
True
```

Lectura, escritura (input, print)

La instrucción `print` permite mostrar información en la pantalla. Como todos los lenguajes de programación, Python tiene un conjunto de reglas preciso que especifica la **sintáxis** (forma) y la **semántica** (significado) de cada instrucción.

En este curso utilizaremos una notación basada en plantillas para ilustrar la sintáxis de algunas instrucciones.

```
print(<expr>, <expr>, ..., <expr>)  
print()  
print(<expr>, <expr>, ..., <expr>, end="\n")
```

Las plantillas anteriores muestran distintas formas de utilizar la instrucción `print`. La primera indica que una instrucción de tipo `print` puede consistir en el nombre de la función `print` seguido de una secuencia de expresiones entre paréntesis, donde las distintas expresiones se separan mediante comas.

La segunda versión muestra que también se puede usar la función `print` sin ninguna expresión.

La notación de ángulos (`<>`) en la plantilla se usa para indicar huecos que se rellenan con otros fragmentos de código Python. El nombre especificado dentro de los ángulos indica el tipo de código que falta; `<expr>` significa una expresión. Los puntos suspensivos indican una sucesión indefinida de expresiones. Los puntos suspensivos no se deben incluir en el código.

Con respecto a la **semántica**, una instrucción `print` muestra información de manera textual. Las expresiones proporcionadas como argumentos se evalúan de izquierda a derecha, y los valores resultantes se escriben en una línea de izquierda a derecha. Por defecto, se inserta un espacio en blanco entre los valores mostrados.

Varias instrucciones de tipo `print` sucesivas suelen escribir sus resultados en líneas separadas en la pantalla. Una instrucción `print` sin parámetros produce la escritura de una línea en blanco en la pantalla.

Instrucción de Asignación

La instrucción de **asignación sencilla** tiene la forma siguiente

$$\langle \textit{variable} \rangle = \langle \textit{expr} \rangle$$

donde *variable* es el identificador de una variable y *expr* es una expresión.

La semántica de la instrucción de asignación consiste en evaluar en primer lugar la expresión de la parte derecha y asociar el valor resultante a la variable de la parte izquierda.

Una variable puede asignarse muchas veces. Esto es, su valor puede cambiar durante la ejecución de un programa. El valor de una variable siempre es último dato que se le ha asignado.

Instrucción de Asignación

En algunos lenguajes de programación una variable es el nombre de una posición (o posiciones) de la memoria del ordenador donde se puede almacenar un valor de un tipo determinado. Cuando se modifica el valor de una variable, el antiguo valor se borra de dicha posición de la memoria y el nuevo valor se escribe allí.

En Python la instrucción de asignación funciona de otro modo. Los valores asignados a una variable pueden estar en cualquier posición de la memoria y las **variables** se utilizan para **hacer referencia a dichos valores** (en realidad a las posiciones de memoria donde se almacenan dichos valores). **Cuando se asigna un valor distinto a una variable**, el valor antiguo no se borra y se sobrescribe con el nuevo valor, simplemente **se hace que la variable pase a referenciar el nuevo valor**.

Instrucción de Asignación

Sin embargo, aunque la asignación no causa directamente el borrado del valor antiguo de una variable, los valores que no son referenciados por ninguna variable se borran de forma automática de manera que el espacio de memoria que ocupan pueda ser utilizado para almacenar nuevos valores. Este proceso se conoce como **recolección de basura** (garbage collection).

Asignación de Input

El propósito de una instrucción de tipo `input` es obtener información del usuario de un programa y almacenar dicha información en una variable. Este proceso se conoce como **lectura de datos**.

En Python, la lectura de datos se realiza utilizando **una instrucción de asignación combinada con** una llamada a la función `input`. Concretamente, la forma de una instrucción de lectura de datos es la siguiente

$$\langle \text{variable} \rangle = \text{input}(\langle \text{prompt} \rangle)$$

donde `<prompt>` es una string que se usa para solicitar al usuario que introduzca la información.

Cuando Python encuentra una llamada a la función `input`, escribe la cadena de caracteres `prompt` en la pantalla y espera a que el usuario teclee un texto y presione la tecla `<Enter>`. A continuación, el texto tecleado por el usuario se almacena en la variable de la parte izquierda de la instrucción de asignación como una string.

Asignación de Input

Cuando el dato que introduce el usuario es un número, es necesario utilizar una forma un poco más complicada de la instrucción de lectura de datos.

```
<variable> = eval(input(<prompt>))
```

En esta instrucción se utiliza la función `eval` para evaluar el resultado de la función `input`. De este modo el texto introducido por el usuario se evalúa como si fuera una expresión para producir como resultado el dato que se almacena en la variable.

Lo importante es recordar que **es necesario evaluar** el resultado de `input` cuando se quiere obtener un número en lugar de una cadena de caracteres.

Si sabemos que el dato introducido por el usuario es un entero (i.e., de tipo `int`), también se puede utilizar la instrucción

```
<variable> = int(input(<prompt>))
```

Asignación múltiple

La **asignación múltiple** es una forma alternativa de la instrucción de asignación que tiene la forma siguiente

$$\langle var \rangle, \dots, \langle var \rangle = \langle expr \rangle, \dots, \langle expr \rangle$$

Semánticamente indica a Python que debe evaluar todas las expresiones de la parte derecha del operador de asignación (=) en primer lugar, y a continuación asignar los valores resultantes de las evaluaciones a las variables de la parte izquierda del operador de asignación.

Debido a que la evaluación de las expresiones de la parte derecha del operador de asignación (=) se realiza en primer lugar, esta instrucción puede utilizarse para intercambiar los valores de dos variables del modo siguiente

$$x, y = y, x$$

Asignación múltiple

También se puede utilizar para obtener varios números del usuario en una sola instrucción.

```
x1, x2 = eval(input("Introduce dos numeros separados por una coma"))
```

La instrucción anterior no funciona para lectura de datos de tipo `string` (que no se evalúa), porque cuando el usuario escribe una coma se interpretará como un caracter más de la entrada. La coma sólo funciona como **separador** de datos cuando la cadena de caracteres se evalúa.

input y print

Asignamos (coloquialmente 'entramos', `input` en inglés) un valor a la variable `x` y luego escribimos (`print`) el resultado.

```
>>> x = int(input("Entra el numero x: "))
Entra el numero x: 5
>>> print("El numero x vale", x)
El numero x vale 5
>>>
```

Descomposición Secuencial

Descomposición Secuencial

Permite resolver un problema descomponiéndolo en una secuencia finita de pasos.

Pasar céntimos a monedas. Dada una cantidad de dinero en céntimos, expresar dicha cantidad en las distintas monedas.

El objetivo es utilizar el **menor número** de monedas posibles. Por ejemplo, 2 monedas de 10 céntimos equivale a una moneda de 20 céntimos.

Apéndice: un poco de historia



- ▶ En 1992 el **Tratado de Maastricht** preveía la creación de la **moneda común** y los **criterios de convergencia**.
- ▶ En 1998 se aprueban los **once países** de la UME a partir de 1999. España está.
- ▶ El **1 de enero de 2002** entran en vigor las monedas y billetes.

Variables (de programación)

- ▶ `c_inicial` almacena el valor inicial de céntimos.
- ▶ Las variables `e2`, `e`, `c50`, `c20`, `c10`, `c5`, `c2`, `c`, almacenarán las cantidades de las distintas monedas.
- ▶ `c_resto` irá almacenando los céntimos que quedan por tratar a lo largo de la descomposición.

Validez: Igualdad

Tras ejecutar la descomposición las variables (los valores almacenados en ellas) han de cumplir:

$$\begin{aligned}c_inicial &= 200 \times e2 + 100 \times e \\ &+ 50 \times c50 + 20 \times c20 + 10 \times c10 \\ &+ 5 \times c5 + 2 \times c2 + c\end{aligned}$$

Validez: Restricciones

Dada la siguiente **división euclídea** (cociente //, resto %):

$$\begin{aligned}c_inicial &= 200 \times e2 + c_resto \\c_resto &= 100 \times e \\ &+ 50 \times c50 + 20 \times c20 + 10 \times c10 \\ &+ 5 \times c5 + 2 \times c2 + c\end{aligned}$$

- ▶ El cociente $e2 = c_inicial // 200$ nos da el número de **monedas de 2 euros**.
- ▶ El $c_resto = c_inicial \% 200$ son los **céntimos que quedan** (el resto de céntimos) tras factorizar las monedas de 2 euros.
- ▶ Se cumple $0 \leq c_resto < 200$. Por lo tanto el número de monedas de 2 euros **es el mayor posible**.

El razonamiento se extiende al resto de las monedas.

Primeros pasos de diseño

Recordemos: para obtener el cociente `//` y para el resto `%`

```
>>> c_inicial = int(input("Entra el numero de centimos: "))
Entra el numero de centimos: 399
>>> c_inicial
399
>>> e2 = c_inicial// 200
>>> e2
1
>>> c_resto = c_inicial % 200
>>> c_resto
199
>>>
```

Usando la división entera podemos seguir calculando el número de monedas. Mirar el siguiente script.

script-pasarcentaMoneda-Seq.py

```
c_inicial = int(input("Entra el numero de centimos: "))
e2 = c_inicial // 200
c_resto = c_inicial % 200
e = c_resto // 100
c_resto = c_resto % 100
c50 = c_resto // 50
c_resto = c_resto % 50
c20 = c_resto // 20
c_resto = c_resto % 20
c10 = c_resto // 10
c_resto = c_resto % 10
c5 = c_resto // 5
c_resto = c_resto % 5
c2 = c_resto // 2
c = c_resto % 2
print("2e:", e2, ",", "e:", e)
print("c50:", c50, ",", "c20:", c20, ",", "c10:", c10)
print("c5:", c5, ",", "c2:", c2, ",", "c:", c)
```

Ejecutando con distintos valores Run->Run Module tenemos:

```
RESTART: ... \script-pasarCentaMonedas-Seq.py
```

```
Entra el numero de centimos: 388
```

```
2e: 1 , e: 1
```

```
c50: 1 , c20: 1 , c10: 1
```

```
c5: 1 , c2: 1 , c: 1
```

```
>>>
```

```
RESTART: ... \script-pasarCentaMonedas-Seq.py
```

```
Entra el numero de centimos: 299
```

```
2e: 1 , e: 0
```

```
c50: 1 , c20: 2 , c10: 0
```

```
c5: 1 , c2: 2 , c: 0
```

Podemos utilizar **asignación múltiple**

Fichero `script-pasarCentaMonedas-Mul.py`:

```
c_inicial = int(input("Entra el numero de centimos: "))
e2, c_resto = c_inicial // 200, c_inicial % 200
e, c_resto = c_resto // 100, c_resto % 100
c50, c_resto = c_resto // 50, c_resto % 50
c20, c_resto = c_resto // 20, c_resto % 20
c10, c_resto = c_resto // 10, c_resto % 10
c5, c_resto = c_resto // 5, c_resto % 5
c2, c = c_resto // 2, c_resto % 2
print("2e:", e2, ",", "e:", e)
print("c50:", c50, ",", "c20:", c20, ",", "c10:", c10)
print("c5:", c5, ",", "c2:", c2, ",", "c:", c)
```

Funciones

Definimos $\mathbb{N} = \{0, 1, 2, 3, \dots\}$

`pasarCentaMonedas` : $\mathbb{N} \rightarrow \mathbb{N} \times \dots \times \mathbb{N}$

`pasarCentaMonedas(20)` = (0, 0, 0, 0, 1, 0, 0, 0)

```
def pasarCentaMonedas(c_inicial):
    e2, c_resto = c_inicial // 200, c_inicial % 200
    e, c_resto = c_resto // 100, c_resto % 100
    c50, c_resto = c_resto // 50, c_resto % 50
    c20, c_resto = c_resto // 20, c_resto % 20
    c10, c_resto = c_resto // 10, c_resto % 10
    c5, c_resto = c_resto // 5, c_resto % 5
    c2, c = c_resto // 2, c_resto % 2
    return e2, e, c50, c20, c10, c5, c2, c
```

Tras compilar `pasarCentaMonedas-Pelada.py` con `Run->Run Module`, llamamos a la función desde el Script

```
>>> pasarCentaMonedas(388)
(1, 1, 1, 1, 1, 1, 1, 1)
>>>
```

Comentario: Notad que **no** aparece `input` ni `print`.

Funciones

Una función es un tipo de subprograma. Los programadores utilizan funciones para reducir la repetición de código y para estructurar y modularizar sus programas.

Una vez que se define una función, se puede llamar múltiples veces y desde varios lugares diferentes en un programa.

Los parámetros permiten que las funciones tengan partes variables.

Los parámetros que aparecen en la definición de una función se denominan **parámetros formales**, y las expresiones que aparecen en la llamada a una función se denominan **parámetros reales** o **argumentos**.

Funciones

La definición de una función en Python tiene la forma siguiente:

```
def <name> (<formal-parameters>):  
    <body>
```

El **nombre de la función** (**name**) debe ser un identificador.

Los **parámetros formales** (**formal-parameters**) consisten en una secuencia (posiblemente vacía) de variables (identificadores) separadas por comas.

El **cuerpo** (**body**) de la función es una secuencia de instrucciones indentadas debajo de la **cabecera de la función**.

Funciones

Los parámetros formales, al igual que todas las variables definidas en el cuerpo de una función (**variables locales**), sólo se pueden utilizar dentro del cuerpo de la función.

En el cuerpo de la función no se puede acceder a variables definidas fuera de la función, aunque tengan el mismo nombre que los parámetros formales o que las variables locales de la función. Es decir, el **ámbito (scope)** de una variable definida dentro de una función o de un parámetro formal de una función es el cuerpo de dicha función.

Funciones

Para usar una función es necesario llamarla (o invocarla) dentro de un programa. Una **llamada a una función** tiene la forma siguiente

```
<name>(<actual-parameters>)
```

donde **name** es el **nombre de la función** y **actual-parameters** es la lista de **parámetros reales** o **argumentos** con los que se pretende usar la función.

Normalmente, cuando la definición de una función tiene varios parámetros formales, los parámetros reales se asocian con los parámetros formales por posición, el primer parámetro real se asigna al primer parámetro formal, el segundo parámetro real al segundo parámetro formal y así sucesivamente.

Funciones

Cuando el intérprete de Python encuentra una instrucción de tipo `return` en el cuerpo de una función

- ▶ termina la ejecución de la función y **devuelve el control** al punto del programa donde se realizó la llamada a la función.
- ▶ Además, el valor o valores de la instrucción `return` se devuelven como el **resultado** de evaluar la expresión formada por la aplicación de la función a los parámetros reales de la llamada particular a dicha función que está siendo evaluada.

Funciones

En Python una función puede devolver más de un resultado. Esto se hace escribiendo varias expresiones separadas por comas en la instrucción `return`. La llamada a una función que devuelve más de un resultado se realiza utilizando una instrucción de `asignación múltiple`.

```
def sumDiff(x, y):  
    return x+y, x-y
```

```
a, b = sumDiff(5,2)  
print("suma_ =_", a, "_diferencia_ =_", b)
```

Como en el caso de los parámetros, cuando una función devuelve varios resultados, la asignación a las variables se realiza por posición. Técnicamente, en Python todas las funciones devuelven un resultado, aunque su cuerpo no contenga una instrucción de tipo `return`. Las funciones que no contienen una instrucción `return` devuelven un objeto especial denominado `None`.

Funciones

Los **valores de retorno** son el mecanismo principal que puede utilizar una función para devolver información a la parte de un programa que realiza una llamada a dicha función.

En algunos casos, las funciones también pueden comunicarse con el programa que las llama realizando **cambios en los parámetros** de la función.

Los **parámetros formales** de una función **sólo reciben los valores de los parámetros reales**.

La función no tiene acceso a las variables correspondientes a los parámetros reales. Por tanto, asignar un valor a un parámetro formal no tiene ningún efecto sobre la variable parámetro real correspondiente.

El tipo de paso de parámetros utilizado por Python se denomina **paso de parámetros por valor**.

Funciones

A pesar de que los parámetros de las funciones de Python se pasan por valor, **si el valor de un parámetro real es un objeto mutable** (por ejemplo, una lista u otro objeto mutable), los cambios que realice la función a dicho objeto serán visibles desde el programa que realizó la llamada a la función.

Esta situación es un ejemplo de un fenómeno conocido en programación como **aliasing**.

main

main

En este curso: Normalmente llamaremos a las funciones desde otras funciones o desde la función `main`.

Mirad y ejecutad el siguiente programa `pasarCentaMonedas-Main.py`

Comentario: Este modo de trabajar recuerda la programación tradicional con `C` o `java`.

pasarCentaMonedas-Main.py

```
def pasarCentaMonedas(c_inicial):
    e2, c_resto = c_inicial // 200, c_inicial % 200
    e, c_resto = c_resto // 100, c_resto % 100
    c50, c_resto = c_resto // 50, c_resto % 50
    c20, c_resto = c_resto // 20, c_resto % 20
    c10, c_resto = c_resto // 10, c_resto % 10
    c5, c_resto = c_resto // 5, c_resto % 5
    c2, c = c_resto // 2, c_resto % 2
    return e2, e, c50, c20, c10, c5, c2, c

if __name__ == "__main__":
    print(pasarCentaMonedas(1))
    print(pasarCentaMonedas(2))
    print(pasarCentaMonedas(4))
    print(pasarCentaMonedas(10))
    print(pasarCentaMonedas(20))
```


El resultado de la ejecución es:

```
>>>
```

```
RESTART: C:...\pasarCentaMonedas-Main.py
```

```
(0, 0, 0, 0, 0, 0, 0, 1)
```

```
(0, 0, 0, 0, 0, 0, 1, 0)
```

```
(0, 0, 0, 0, 0, 0, 2, 0)
```

```
(0, 0, 0, 0, 1, 0, 0, 0)
```

```
(0, 0, 0, 1, 0, 0, 0, 0)
```

```
>>>
```

Validar con Doctest

Validar con doctest

Se añade un `doctest` que permita `validar` el funcionamiento de la función.

La siguiente transparencia corresponde a `pasarCentaMonedas-Doctest.py`

Más detalles en el laboratorio.

```

def pasarCentaMonedas(c_inicial):
    """
    >>> pasarCentaMonedas(1)
    (0, 0, 0, 0, 0, 0, 0, 0, 1)
    >>> pasarCentaMonedas(2)
    (0, 0, 0, 0, 0, 0, 0, 1, 0)
    >>> pasarCentaMonedas(5)
    (0, 0, 0, 0, 0, 1, 0, 0, 0)
    >>> pasarCentaMonedas(10)
    (0, 0, 0, 0, 1, 0, 0, 0, 0)
    >>> pasarCentaMonedas(20)
    (0, 0, 0, 1, 0, 0, 0, 0, 0)
    """

    e2, c_resto = c_inicial // 200, c_inicial % 200
    e, c_resto = c_resto // 100, c_resto % 100
    c50, c_resto = c_resto // 50, c_resto % 50
    c20, c_resto = c_resto // 20, c_resto % 20
    c10, c_resto = c_resto // 10, c_resto % 10
    c5, c_resto = c_resto // 5, c_resto % 5
    c2, c = c_resto // 2, c_resto % 2
    return e2, e, c50, c20, c10, c5, c2, c

if __name__ == "__main__":
    import doctest
    doctest.testmod()

```

Versión doctest.testmod()

Al ejecutar Run->Run Module comprobamos que pasa el juego de pruebas:

```
>>>  
RESTART: ... pasarCentaMonedas-Doctest.py  
>>>
```

Mirar en el laboratorio qué ocurre cuando no supera algún juego de pruebas

Versión doctest.testmod(verbose = True)

Cuando se ejecuta pasarCentaMonedas-Verbose.py:

```
>>>
RESTART: ... \pasarCentaMonedas-Verbose.py
Trying:
    pasarCentaMonedas(1)
Expecting:
    (0, 0, 0, 0, 0, 0, 0, 1)
ok
Trying:
    pasarCentaMonedas(2)
Expecting:
    (0, 0, 0, 0, 0, 0, 1, 0)
ok
Trying:
    pasarCentaMonedas(5)
Expecting:
    (0, 0, 0, 0, 0, 0, 1, 0, 0)
ok
...
5 passed and 0 failed.
Test passed.
>>>
```

Comentario: En main es posible incluir a la vez `prints` y `doctest`.

Introducción a `if`

Normalmente pensamos que los ordenadores ejecutan una serie de instrucciones en orden secuencial.

La **composición secuencial** es un concepto fundamental en la programación, pero no permite resolver todos los problemas.

Las instrucciones condicionales (e.g., `if`) permiten que un programa ejecute diferentes secuencias de instrucciones en distintos casos, de manera que el programa pueda elegir el tipo de acciones adecuadas a cada situación. Concretamente, las instrucciones de tipo **`if-elif-else`** que veremos en esta sección se denominan **estructuras de decisión**.

Un aspecto interesante de las instrucciones condicionales (como `if`) es que permiten **alterar el flujo de control** de un programa, eligiendo las secuencias de acciones deben ejecutarse y las que no deben ejecutarse del código del programa. Por este motivo se denominan **estructuras de control**.

Introducción a if

Los **diagramas de flujo** se utilizan para describir gráficamente la semántica de ciertas estructuras de control.

Un diagrama de flujo contiene **cajas** que representan distintas partes de un programa y **flechas** que van de unas cajas a otras indicando la secuencia de eventos que tienen lugar durante una ejecución del programa.

Las cajas con forma de diamante muestran decisiones condicionales. Si una condición es cierta, el control se transfiere a la caja asociada a dicha condición, que contiene la secuencia de instrucciones a realizar en dicho caso. Si la condición es falsa, el control pasa a la siguiente condición de la estructura de decisión, si existe dicha condición, o a la siguiente instrucción en el programa, en otro caso.

Una vez que las instrucciones de la secuencia asociada a la primera condición cierta han terminado de ejecutarse, el control pasa a la instrucción que sigue a la estructura de decisión en el código del programa.

Introducción a if

Una instrucción de tipo `if` tiene la siguiente forma

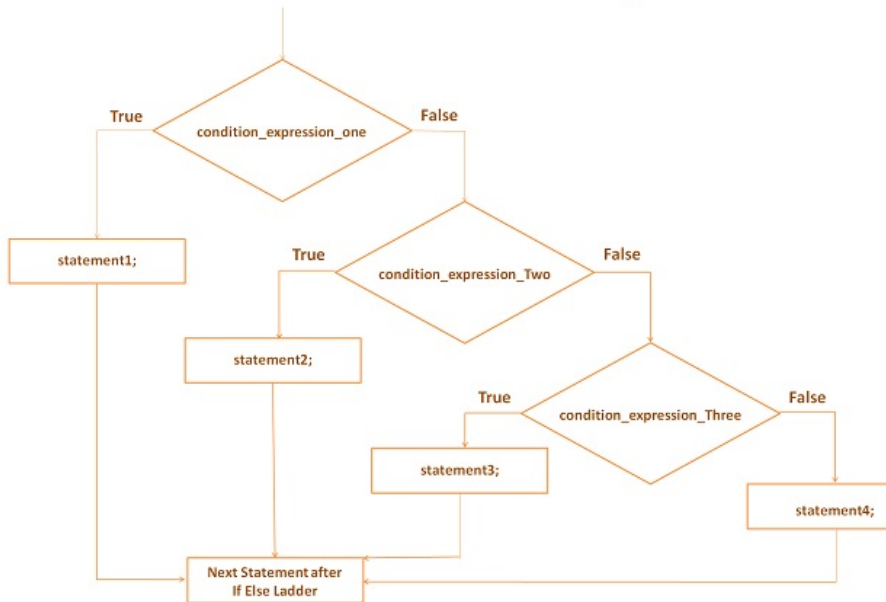
```
if <condition>:  
    <body>
```

El **cuerpo** (`body`) es una secuencia formada por una o más instrucciones **indentadas** debajo de la **cabecera de if**.

La semántica de `if` consiste en evaluar en primer lugar la **condición** de la cabecera. Si la condición es cierta, se ejecuta la secuencia de instrucciones del **cuerpo**. En otro caso, las instrucciones del cuerpo no se ejecutan.

Este tipo de instrucción se denomina **decisión simple** o de **una sola rama**.

If Else Ladder Statement Flow Diagram



Decisión con dos alternativas

Una estructura de decisión con **dos alternativas** o dos ramas tiene la forma siguiente

```
if <condition>:  
    <statements>  
else:  
    <statements>
```

Cuando el intérprete de Python encuentra esta estructura, evalúa la **condición** en primer lugar. Si la condición es cierta, ejecuta las instrucciones indentadas debajo de la **parte if**; en otro caso, ejecuta las instrucciones indentadas debajo de la **parte else**. A continuación, el control pasa a la instrucción que sigue a la estructura de decisión (**if-else**) en el código del programa.

Decisión con múltiples alternativas

Una estructura de decisión con **múltiples alternativas** o ramas tiene la forma siguiente

```
if <condition1>:  
    <case1 statements>  
elif <condition2>:  
    <case2 statements>  
...  
else:  
    <default statements>
```

Esta estructura se utiliza para combinar varios bloques de código mutuamente excluyentes.

Instrucción `if`

El intérprete de Python evalúa las condiciones de arriba abajo hasta que encuentra la primera condición que es cierta.

Si encuentra una condición que sea cierta, evalúa las instrucciones indentadas debajo de dicha condición, y a continuación pasa el control a la instrucción que sigue a la estructura de decisión (`if-elif-else`) en el código del programa.

Si ninguna de las condiciones es cierta, evalúa las instrucciones indentadas debajo de la parte `else`.

La cláusula `else` es opcional. Si se omite, es posible que no se ejecute ninguna instrucción de la estructura de decisión.

Instrucción if

La instrucción `if` permite tratar distintos casos.

El siguiente script (almacenado en `script-if.py`) da un ejemplo de la intruccion alternativa `if`:

```
x = int(input("Please enter an integer: "))
if x < 0:
    x = 0
    print('Negative changed to zero')
elif x == 0:
    print('Zero')
elif x == 1:
    print('Single')
else:
    print('More')
```

Ejemplo: ordenar 3 enteros

Algoritmo basado en permutaciones: Recordad que dados 3 enteros, por ejemplo x, y, z se pueden **ordenar** de $3 \times 2 = 6$ maneras distintas:

x, y, z

x, z, y

y, x, z

y, z, x

z, x, y

z, y, x

Supongamos $x=5, y=2, z=8$. Se ordenan y, x, z y corresponde al caso $y \leq x \leq z$.

Esto nos da la función:

```
def sort3perm(x,y,z):  
    if x<=y<=z : return (x,y,z)  
    if x<=z<=y : return (x,z,y)  
    if y<=x<=z : return (y,x,z)  
    if y<=z<=x : return (y,z,x)  
    if z<=x<=y : return (z,x,y)  
    return z, y, x
```

Almacenada en sort3perm-pelada.py

El doctest sugerido es:

```
def sort3perm(x,y,z):  
    '''  
    >>> sort3perm(1,2,3)  
    (1, 2, 3)  
    >>> sort3perm(1,3,2)  
    (1, 2, 3)  
    >>> sort3perm(2,1,3)  
    (1, 2, 3)  
    >>> sort3perm(2,3,1)  
    (1, 2, 3)  
    >>> sort3perm(3,1,2)  
    (1, 2, 3)  
    >>> sort3perm(3,2,1)  
    (1, 2, 3)  
    '''  
    ...
```

La función completa se encuentra en `sort3perm.py`

Algoritmo basado en ordenes parciales: Las variables primero, segundo se usan para para mantener un orden parcial. Tratamos x, y, z uno tras otro ampliando el orden.

En el nombre `sort3FS` la **F** viene de first y la **S** de second.

```
def sort3FS(x,y,z):
    first=x
    if y <= first:
        first, second = y, first
    else:
        second =y
    if z <= first:
        return z, first, second
    elif first < z <= second:
        return first, z, second
    else:
        return first, second, z
```

La función se encuentra en `sort3FS-pelada.py`

Apéndice para apasionados: sobre genericidad

En `sort3perm(x,y,z)` y `sort3FS(x,y,z)`

- ▶ Nunca hemos dicho que `x`, `y` y `z` tuvieran que ser `int`.
- ▶ De hecho, solo hace falta que `"<"` i `"<="` esten definidos

Como la clase `srt` los tiene definidos, entonces

```
>>> "hola"<"adios"  
False  
>>> "hola"<="hola"  
True  
>>> sort3FS("primavera", "verano", "otoño")  
( 'otoño', 'primavera', 'verano' )
```

Llamamos a estas funciones [genéricas](#).

Introducción a for

Los programadores utilizan **bucles** para ejecutar una secuencia de instrucciones varias veces. El tipo más sencillo de bucle es un **bucle definido**. En este tipo de bucle, en el momento en el que comienza el bucle se sabe cuantas veces **iterará** (repetirá) **el cuerpo del bucle** (la secuencia de instrucciones).

Un bucle en Python tiene la forma siguiente

```
for <var> in <sequence>:  
    <body>
```

El cuerpo (**body**) del bucle puede ser cualquier secuencia de instrucciones. El ámbito del cuerpo de un bucle (i.e., su extensión) se indica mediante su **indentación** con respecto a la **cabecera del bucle** (la parte `for <var> in <sequence>:`).

La variable que aparece después de la palabra clave **for** se denomina el **índice** del bucle. En cada iteración dicha variable toma uno de los valores sucesivos de la **secuencia** (sequence) y ejecuta las instrucciones del cuerpo del bucle para dicho valor.

Introducción a for

A menudo la secuencia consiste en una **lista** de valores. Las listas son un concepto muy importante en Python que estudiaremos en detalle más adelante. De momento basta saber que se puede crear una lista encerrando una secuencia de valores entre corchetes.

```
for x in [1, 3, 5, 7]:  
    print(x*x)
```

La función **range** de Python genera una secuencia de números en el momento de la ejecución. Por ejemplo, la llamada **range(10)** genera la secuencia de números 0, 1, 2, ..., 9. En general, **range(<expr>)** produce una secuencia de números consecutivos que comienza en 0 y termina en $\langle \text{expr} \rangle - 1$.

El valor de $\langle \text{expr} \rangle$ determina, por tanto, el número de elementos de la secuencia resultante. El siguiente bucle, que utiliza **range** y una variable como **contador**, es muy común.

```
for x in range(10):  
    print(x)
```

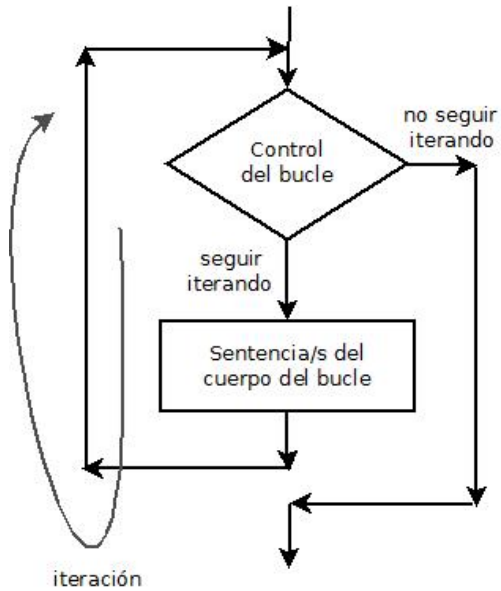
Introducción a for

Normalmente pensamos que los ordenadores ejecutan una serie de instrucciones en orden secuencial. Lo interesante de **los bucles** es que **alteran** el **flujo de control** del programa.

La introducción de un bucle hace que el programa vuelva atrás y ejecute ciertas instrucciones una y otra vez.

Instrucciones como **for** se denominan **estructuras de control**, porque controlan la ejecución de otras partes del programa. Concretamente, las instrucciones de tipo **for** se denominan **estructuras iterativas**.

Los **diagramas de flujo** se utilizan para describir gráficamente la semántica de ciertas estructuras de control. Un diagrama de flujo contiene **cajas** que representan distintas partes de un programa y **flechas** que van de unas cajas a otras indicando la secuencia de eventos que tienen lugar durante una ejecución del programa.



El For de Python

- ▶ El `for` de Python permite **iterar a través de los elementos de una secuencia**.
- ▶ Difiere un poco del de C o Pascal.

La función range()

```
>>> list(range(5))  
[0, 1, 2, 3, 4]
```

Atención: Hay 5 valores que son 0, 1, 2, 3, 4. Notad que el 5 no aparece en la lista.

Mi primer for

Si se necesita iterar sobre una secuencia de números, la función `range()` es muy útil.

```
>>> for i in range(5):  
print(i)
```

```
0  
1  
2  
3  
4  
>>>
```

La siguiente función `contar(n)`, dado `n` imprime los números de 1 a `n`. Nótese que en esta función no hay `return`.

El código se encuentra en `contar.py`

```
def contar(n):
    """
    >>> contar(0)
    >>> contar(10)
    1
    2
    3
    4
    5
    6
    7
    8
    9
    10
    """
    for x in range(1, n + 1):
        print(x)
if __name__ == "__main__":
    contar(0)
    contar(10)
    import doctest
    doctest.testmod()
```

Ejemplo: tabla de multiplicar

```
def tabla_multiplicar(n):
    """
    >>> tabla_multiplicar(10)
    10 * 1 = 10
    10 * 2 = 20
    10 * 3 = 30
    10 * 4 = 40
    10 * 5 = 50
    10 * 6 = 60
    10 * 7 = 70
    10 * 8 = 80
    10 * 9 = 90
    10 * 10 = 100
    """
    for x in range(1, 11):
        print(n, '*', x, '=', n * x)
if __name__ == "__main__":
    tabla_multiplicar(10)
    import doctest
    doctest.testmod()
```

Código en `tabla_multiplicar.py`